CISCO SYSTEMS

Document Number	ENG-136393
Revision	0.5
Author	Doug Wooff
Project Manager	Mala Devlin

Project Name:

HFR MBI Manager Functional Specification

Project Headline

One-line description of the project. Include the keywords specific to this program/project that will make it easy for someone to identify what this document is about.

Reviewers

Department	Name
Project Manager	Name (Title)
Engineering	Name (Title)
Product Marketing	Name (Title)
Customer Advocacy	Name (Title)
Tech Pubs	Name (Title)
Test	Name (Title)
Quality	Name (Title)

Modification History

Rev	Date	Originator	Comment
0.1	[redacted]	Doug Wooff	Initial skeleton
0.2	[redacted]	Doug Wooff	Add figure, Install Management background.
0.3	[redacted]	Doug Wooff	Add install examples (from GSR) for small prereview.
0.4	[redacted]	Doug Wooff	Add CLI, test considerations, incorporate feedback
0.5	[redacted]	Doug Wooff	Add mbi-hello context as an appendix

EXHIBIT 1.3

Table of Contents

Add the table of contents when the first draft is complete. The necessary paragraph formats are included.

Definitions

MBI	Minimum Boot Image
SC	Shelf Controller (used for 'SCRP' in this document)
RP	Route Processor
SCRP	Shelf Controller Route Processor: handles local shelf (aka "rack") management jobs as well as some routing work (RP).
DRP	Distributed Route Processor
Platform Manager	Software entity on the SC. Its "umbrella" includes hardware discovery, OIR, and PCtl.
PCtl	Plugin Controller: performs the card state management and software loading functionalities for all non-SC cards within a rack.
dSC	designated Shelf Controller
dSCp	designated Shelf Controller (election) process
LR	Logical Router
ROMMON	ROM Monitor
SP	Service Processor
Qnet	Inter-node networking software.
QSM	Qnet Symlink Manager
GSP	Group Services Protocol
InstMgr	Install Manager (also InstallMgr)
InstHlpr	Install Helper (also insthelper)
Install stream	The set of "ENA component files", contained in one or more directories on a mass storage device, corresponding to a software release plus its upgrades, targetted towards a particular platform.

1.0 Introduction

An MBI is a Minimum Boot Image, a subset of ENA components which can be launched and executed independently on a CPU. It is normally launched in preparation for loading a follow-on set of associated ENA components, and is tailored to suit the hardware of the platform on which it executes.

The MBI Manager (MbiMgr) is a software component residing on each SC or SCRP in the HFR system. It is an interface between clients interested in booting a card's MBI, and the Install Manager, who has pre-assigned and stored an MBI for that card.

A list of components residing in the MBI can be found in section 3.2

1.1 Purpose and Problem Definition

The main purpose of MbiMgr is to simplify and centralize the task of validating an MBI file specification ("filespec") as given by a client. Alternatively, MbiMgr might fetch a pointer to another, more appropriate MBI to boot for a given "node".

Note that "node" in this discussion refers to an independent CPU implementing an "ENA node"; there may be several nodes on an actual, physical card.

There are two currently known MbiMgr clients. One is 'dSCp' [1] which needs to validate MBI's for booting SC's, and 'Plugin Controller' [2] which needs to validate MBI's for all other cards.

In each case, the client has received a 'BOOT_REQUEST' message from a node's rommon Inside the message, whose format is defined in [3], is contained (possibly) a list of MBI's which are stored in the node's bootflash - as discovered by rommon. The list could be empty if no MBI files are discovered.

The client will use the data in the BOOT_REQUEST message to call an MbiMgr API (the "validate" API) in an attempt to have one of the given MBI's validated as 'an acceptable MBI to boot'.

MbiMgr responds in its Validate API with one of three possibilities:

1. one of the client's indicated MBI's (it is echoed back as 'acceptable')

2. a file specification for a different MBI, so that it can be 'tftp booted' by rommon

3. a NULL filespec, indicating that no MBI has been authorized for this node

For case (2), all required tftp server information for rommon to be able to perform the tftp boot is also returned to the client.

For case (3), the node should be properly put to sleep - if possible - or ignored, with the appropriate user notification of the error condition.

1.2 Functions

MbiMgr provides a "validation" API for clients to check whether known, available MBI's are acceptable, or if some other externally-resident MBI needs to be launched.

The decision for selecting an MBI is based on the past history of issuing "install"-type commands, and possibly LR assignments. Currently, all of this information is envisioned to be contained within or accessed by the Install Manager.

MbiMgr provides a single point of access to the relevant InstallMgr data.

As a tester and developer convenience, a CLI "override" to this normal customer-oriented approach to launching software images is also provided. A tester can, via configuration command, point to a full .vm image file stored in various (ie. flexible) locations. Upon the next node reload, the indicated .vm file will be launched in place of the default MBI for that node.

1.3 Audience

The intended audience for the document is any person working on HFR who:

- develops software concerning image management

- launches customized images on a routine basis

- performs testing on system or card bringup

2.0 Functional Overview

2.1 System Overview

After booting a node's rommon, launching an MBI is the essential next step in loading and launching the software components that have been activated for a particular node. In general, only once all of or a significant portion of the complete set of components "assembled" for a node have been loaded can the node do any useful work.

Launching an MBI on its own is not a very interesting or useful achievement. If software components have also been pre-assigned (ie. "installed and activated") for the card, in association with an MBI, then they too will be loaded, and the node can perform useful work.

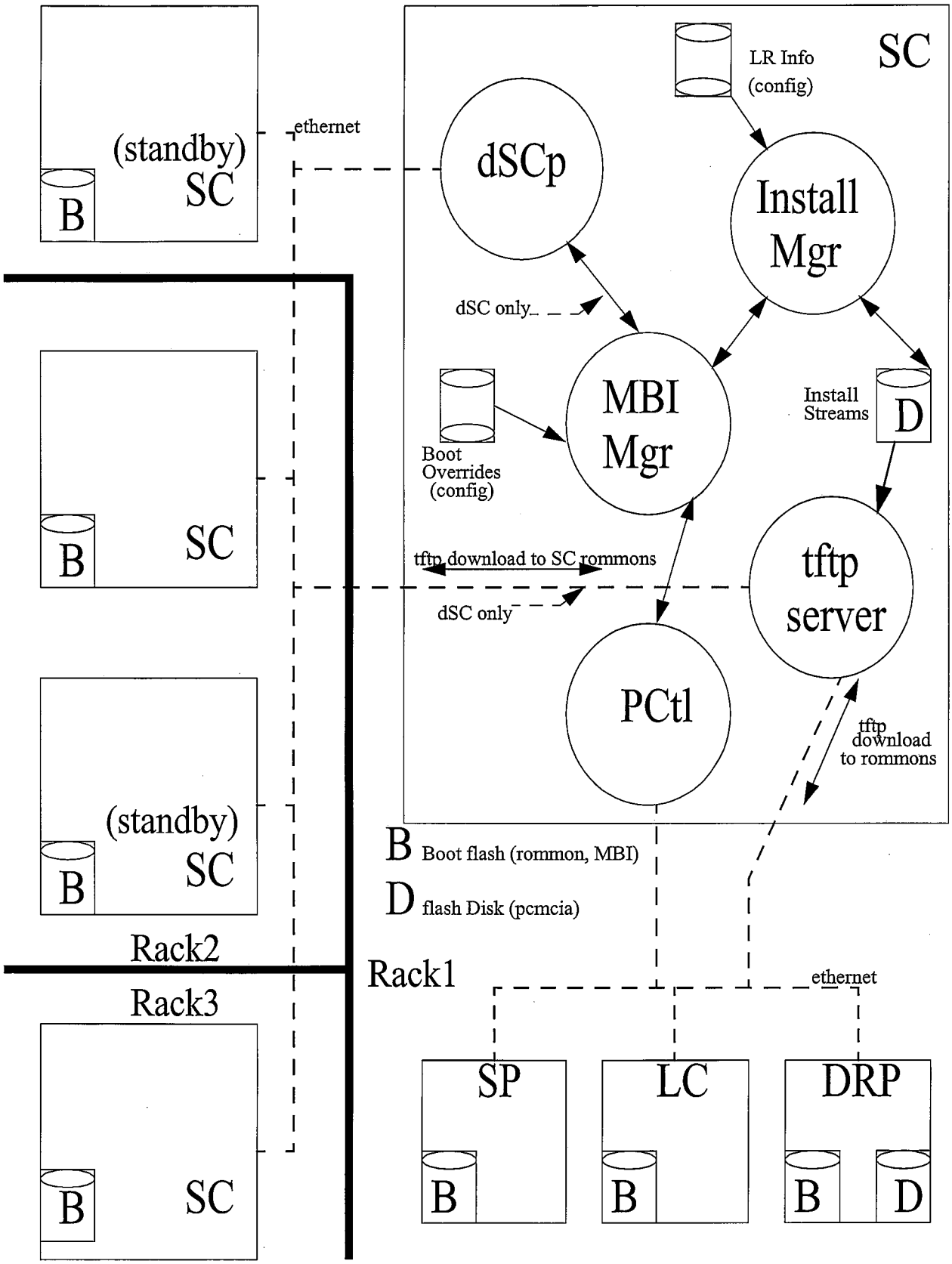
One of the MBI's key components, the Install Helper, knows how to locate the appropriate pre-installed software components for the card, ie. the "install stream", that has been activated for that node. Install Helper may examine the local 'ldpath' file, and will communicate with the InstallMgr running on a higher authority node. If no components have been pre-assigned with a user's "install accept" command, and then activated, then an arbitrary MBI will basically just "sit there", doing nothing except creating an ENA node. An MBI "floating alone" like this is envisioned to be useful only for developer debug activities.

Since a node's rommon may discover more than one MBI stored in its local bootflash, the client which is managing the node needs to be able to choose an MBI to boot. MbiMgr provides a common interface between clients and the Install Manager to make this choice.

If the booting node is an SC, and the booting rommon does *not* hear from a dSC in order to validate one of its MBI's, then rommon will time out and pick one MBI to boot. There is currently no rule to follow to decide which one to pick. If an MBI was validated in the past, however, then its filespec will be stored as a rommon variable, and *it* should be chosen in the absence of a validating dSC. Otherwise, if no such rommon variable exists, then one of the MBI's available must be chosen in the case of a timeout. [Issue #1].

MbiMgr provides a "boot override" configuration command. These test commands, detailed in the CLI section 7.0, "End User Interface", allow a tester to point to any arbitrary self-contained image (aka "full .vm") or even another MBI if desired. As previously indicated, launching a new unassociated MBI establishes an ENA qnet node, but will do little else of interest. The 'boot override' configuration information is shown in Fig. 1 as an input to MbiMgr.

The following diagram, Fig. 1, shows the context in which MbiMgr operates. This context diagram is intended to convey information flow between software entities. Two separate (internal) ethernet networks are shown for the purposes of clarity, showing the paths taken by certain message types. In reality, they are both the same network.


SC
LR Info
(config)
dSCp
Install
Mgr
ethernet
(standby)
SC
B
dSC only
MBI
Mgr
Install
Streams
D
Boot
Overrides
(config)
SC
B
tftp download to SC rommons
dSC only
tftp
server
PCtl
tftp
download
to rommons
(standby)
SC
B
B Boot flash (rommon, MBI)
D flash Disk (pcmcia)
Rack2
Rack1
ethernet
Rack3
SP
LC
DRP
SC
B
B
B
B
D

The large 'B' indicates the Bootflash storage which is local to each CPU in the system. It stores rommon, one or more versions of an MBI (as ordinary files), and the 'ldpath' file used by InstHlpr. There is some extra reserved space on each bootflash, but there may not be enough, in general, to store a node's entire install stream (ie. all of its ENA components).

If there is enough bootflash space available, the install stream *can* be completely stored locally, which could facilitate faster node reboot. This possibility has been envisioned for line cards, for example.

Another proposed usage for bootflash is to cache a local copy of the configuration data applicable to that node.

Note that, as MBI and ldpath (and possibly other) files are written over time, and older files are marked as "deleted", a squeeze operation may eventually be required in order to free up space for a new upgrade. This problem will, at a later date, be handled by using a new and enhanced flash file system which performs the squeeze operation in the background.

The large 'D' indicates where larger mass storage devices such as flash Disks are available. The flash disk (pcmcia card) on the SC stores components for all nodes. DRP cards also have local mass storage, so some files can be off-loaded from the SC.

MbiMgr runs on every SC. On a standby SC, it is essentially sleeping, since it will never be consulted. It will be consulted about **SC** MBI's only if it is running on the dSC.

It can be inferred from Fig. 1 that PCtl manages cards inside a rack, except the SC's themselves. (PCtl running on a primary SC might perform some ligtweight monitoring of a standby SC). All of these (non-SC) cards within a rack, when their nodes' rommon boots, will send BOOT_REQEST messages to PCtl. PCtl will use the provided information, and call the MbiMgr's "validate" API.

MbiMgr confers with InstMgr to see if an indicated MBI was previously installed (using 'install accept') for this node type and within its LR. If so, the user would have issued an 'install accept' command to InstallMgr, then appropriate files would have been copied to mass storage (including the MBI), and the "master" ldpath file updated.

At this time, after the 'install accept' operation, independent of whether the new install stream is activated or not, InstallMgr must update the bootflash of all implicated nodes.

InstallMgr must ensure that the new MBI is copied to the affected nodes, and that the local 'ldpath' file is updated. This operation is just to make the new MBI *available* for activation. Note that a local ldpath file contains information pertinent only to that node. This method - updating the bootflash for all accepted install streams - provides a simple method to subsequently *activate* any one of them.

In fact, once accepted. a user could quite easily switch back and forth between install streams (and their associated MBI's) simply by activating one or another. This will likely form a large part of the upgrade/downgrade strategy.

In a similar way, at a higher level, when an SC rommon boots, it sends BOOT_REQUEST messages to the dSCp (dSC election and management process) resident on the SC. dSCp runs on each SC, but will only look at BOOT_REQUEST messages if it is running on the dSC. dSCp will use the provided information, and call the MbiMgr's "validate" API.

MbiMgr confers with InstallMgr to see if an indicated MBI was previously installed (using 'install accept') for the SC card type and within its LR.

Even though SC's may be considered part of the HFR owner plane, and must run compatible software, they will also be associated with user LR's. It is conceivable and likely that users may update a given SC, and not desire to have a new feature immediately activated on all SC's. Similarly, activating an install stream on a given SC should not automatically update all running SC's.

MbiMgr will not be consulted, ie. does not *need* to be consulted, by dSCp running on an "ordinary" (ie. non-dSC) SC, since only a dSC will serve boot requests from other SC's. Similarly, the tftp server will never be used to serve other SC's on an ordinary SC.

From this discussion, it can be concluded that InstallMgr on *each* SC needs to have access to all accepted install streams. The current plan calls for the replication of the files on each SC. This way, any SC can handle all cards within its rack. Also, any SC could, in theory, become dSC, and thereby be ready to serve all other SC's. So every SC needs a local copy of all possible SC files (as well as files for other card types).

From Fig. 1, we note that there is no direct interaction between MbiMgr, its clients, nor InstallMgr, with the tftp server. The tftp server "floats" as a well-known service, ready to serve tftp client requests directly from a node's rommon. tftp server needs only be able to read any MBI images from the mass storage device previously written to by InstallMgr.

The BOOT_REQUEST and BOOT_REPLY message formats are documented in the Card Configuration Protocol (CCP) document [3]. Message exchange timelines therein better describe the bootup event timings.

2.2 Features

In addition to validating MBI's for booting which have been "properly" installed, MbiMgr will provide a "back door" to download any desired bootable .vm file to any card on the system via tftp. This feature is desired by manufacturing, testing, and possibly developers.

MbiMgr will essentially "trick" the calling client and point to the desired .vm file instead of an MBI, when so configured. The client will then tell rommon to tftp the indicated file and launch it. rommon doesn't know or care what is inside the indicated file. Normally, if this override is used, a "full image" file will be indicated so that it can do useful work.

A "full image", also known as a "full .vm", is a self-contained monolithic collection of all ENA components, including an MBI, required to run applications on a given card type. It is loaded directly into CPU ram and executed from there.

Another test feature will allow a user to force rommon to ignore any MBI's on a node's bootflash, and have the MBI tftp transferred regardless. This allows the important "fallback" path to be exercised.

The MBI filenames will be defined at build time, and the file *name* itself will ***not*** have a version number. When the Boot Package (BootPkg) to which an MBI belongs is installed (along with the install stream's component files), it is placed in a *directory* with a name which includes a version number. This makes a given MBI file unique in a filesystem. A copy of the installed MBI is ***also*** made to a node's bootflash when the BootPkg is installed for that node, using the same directory name. The combination of directory name (versioned) plus MBI file name (unversioned) gives the MBI a unique filespec. The storage device to which it was originally installed may also become part of its identifier.

A user/tester can configure which software release (install stream, including its associated MBI) to boot for any given node by following the official installation procedures. More than one install stream can be created (ie. installed) for any given node, but only one can be ***activated***.

For each install stream which is 'install accept'ed for a node, the corresponding MBI is copied to the node's bootflash, and an entry added to the local 'ldpath' file. Which MBI gets used to boot depends on which install stream gets activated for that node.

Booting a particular MBI then implies using its associated install stream.

Note that it may be considerd superfluous to cache a copy of the 'ldpath' file on bootflash for some cards, such as SP's and LC's. This file must be validated against the master copy of the SC in any case.

Initially, any installed release within a given LR will have a default MBI and associated component files per card-type. A user can then override that for any desired card-type or node per LR using InstallMgr.

3.0 Design Considerations

In order to better understand how MbiMgr achieves its goals, it is useful to understand the environment in which it operates. This section will present a brief overview of how image sets (ie. install streams) are manipulated. The existing situation using a GSR as an example will be discussd, then extrapolated to HFR.

For a full and complete description, the "Install Management" documents should be referenced. [4] [5] [6].

It can be considered that the MBI (ie. its Boot Package) exist simply to provide qnet connectivity for Install Helper. Once done, InstHlpr contacts InstallMgr at a well-known location, and loads in the appropriate ENA components for the node.

The location needs to be "well-known" since QSM will not be available in the MBI image. QSM requires GSP, and it is desireable to keep these two components out of the MBI. The MBI complexity is thereby reduced, and future upgrades to QSM and GSP are simplified.

Another reason that the path to the InstallMgr server must be well-known is that qnet will no longer be sending node-alive broadcasts. A client must simply try to open a connection to a known destination using a well-known pathname. The location of InstallMgr is left in SYSPAGE by rommon for this purpose [Issue 2]. This is known as the "boot server source". At the time rommon's BOOT_REQUEST was answered, it was from a known source, which simply needs to be recorded for the MBI.

At this time, there are no known users of this boot server source data except Install Helper. All other components in the MBI are dedicated to providing qnet communication and basic infrastructure services.

3.1 GSR Installation Example

The life of software on a GSR begins with a "starter image", a "full .vm" which contains all components to be needed by the system in a monolithic file. At the RP rommon prompt, a user boots (using tftp, here) the image:

```
rommon> boot dwooff/ena12000.vm
```

Once it settles,

```
install accept mem: to disk0:
```

copies all components to appropriate directories on disk0: (a PCMCIA flash disk, in this example). The user then does a 'reload' back to rommon and

```
rommon> boot disk0:12k-rp-1.0.0/enaboot-grp.vm
```

boots the "MBI" associated with the "install stream" just placed on disk0:.

The 'boot image' (enaboot-grp.vm) is the MBI equivalent for GSR. It launches, then install management components access the 'ldpath' file to see where to load the rest of the ENA components. 'ldpath' is stored near the root (under / instdb) on the device on which the code was installed (here, "disk0:").

```
ios#dir disk0:
Directory of disk0:
4       drwx  4096   [redacted]12k-rp-1.0.0
```

11910 drwx 4096 [redacted] instdb
65696 -rwx 4923 [redacted] sam_certdb
65760 -rwx 141 [redacted] sam_crldb
ios#dir disk0:/instdb
ldpath

ios#run cat disk0:/instdb/ldpath
/disk0/12k-rp-1.0.0:/ /disk0/12k-rp-1.0.0/enaboot-grp.vm

Note that there is an association of a directory (first item) and the MBI (final item). When we boot an MBI, it looks inside 'ldpath' to find mention of itself. In front of that entry are the ordered LOADPATH directories to be used.

Suppose now we'd like to upgrade a component - on the fly, of course, without reloading the RP. We use the 'install add' command:

install add tftp://223.255.254.254/dwooff/platforms__c12000__chassis-control__sysldr-rp.pie to disk0: start

This command fetches the 'pie' file, breaks it up into components (here, there is only one) and copies them to new directories on the indicated device, disk0:.

ios#dir disk0:
Directory of disk0:
4 drwx 4096 [redacted] 12k-rp-1.0.0
11910 drwx 4096 [redacted] instdb
65696 -rwx 4923 [redacted] sam_certdb
65760 -rwx 141 [redacted] sam_crldb
11916 drwx 4096 [redacted] c12000-sysldr-rp-0.0.0

The latter directory has been added. The 'start' option of the 'install add' command forces all servers inside the installed package to restart, using the new files. LOADPATH will have already been updated before the restart, so all new references to files in the newly installed component(s) will find the new files first. The corresponding original files will henceforth be ignored. 'ldpath' now looks like this:

ios#run cat disk0:/instdb/ldpath
/disk0/c12000-sysldr-rp-0.0.0:/disk0/12k-rp-1.0.0:/ /disk0/12k-rp-1.0.0/enaboot-grp.vm

To demonstrate how two MBI's and their corresponding install streams live side by side, we reload the RP, and boot a new starter .vm:

rommon> boot dwooff/ena12000-0.10.7.vm

then

install accept mem: to disk0:

as before for the first install stream. After the operation finishes (a lot of disk space is required!), the new 'ldpath' file with 2 install stream entries looks like this:

ios#run cat disk0:/instdb/ldpath

/disk0/c12000-sysldr-rp-0.0.0:/disk0/12k-rp-1.0.0:/ /disk0/12k-rp-1.0.0/enaboot-grp.vm

/disk0/12k-rp-0.10.7:/ /disk0/12k-rp-0.10.7/enaboot-grp.vm

An entry for the new MBI and its associated install stream has simply been added to the file.

The disk now looks as follows:

ios#dir disk0:

Directory of disk0:

4	drwx	4096	[redacted] 12k-rp-1.0.0
11910	drwx	4096	[redacted] instdb
65696	-rwx	4923	[redacted] sam_certdb
65760	-rwx	141	[redacted] sam_crldb
11916	drwx	4096	[redacted] c12000-sysldr-rp-0.0.0
12062	drwx	4096	[redacted] 12000-rp-all-0.10.7

In order to select one install stream or the other, the user simply boots an MBI from one directory or the other. Install management intelligence inside the MBI then searches 'ldpath', finds the LOADPATH for itself, and loads and launches all of the corresponding ENA components.

In HFR, this style of file management will be done for every node; ie. every SP, LC, SC, and DRP's SMP complex. Each "ENA node" has its own bootflash, which stores its rommon, one or more MBI's, and the ldpath file. Where space permits, a card's install stream can also be stored on bootflash for faster boot times.

The MBI's and ldpath are ***copies*** of the real, main files kept on the mass storage device which accepted the original installation. They are kept here strictly as an optimized boot time convenience. If a "local" MBI cannot be launched for any reason, a new one will be tftp'd by rommon from a higher level server.

Either way, the MBI runs, and looks for the local ldpath file. If the local file cannot be read for any reason, then Install Helper will consult with InstallMgr via qnet to get its LOADPATH, and continue to load files.

3.2 MBI Component List

The set of components which constitute an MBI and are placed in the Boot Package are as follows.

- microkernel, dllmgr, platform startup code, infrastructure (LWM, EM, etc)
- system manager
- control plane ethernet server
- bootflash driver
- Qnet
- dumper
- MBI-Hello process
- kosh shell
- syslog helper
- Config Helper (TDB)
- Install Helper
- Client libraries for Install Helper [Issue #5]
- control plane switch driver (SP/SC only)
- serial driver (con/aux) (SC only)
- PCMCIA flash disk driver (SC/DRP only)

4.0 External Interface

There is one main "validate" API which a client can call. Other API's for configuring temporary download images will also be defined.

4.1 The Validate API

Some structure types are needed to use the API:

(some types for these fields may be pre-defined, ie. not 'uint32')

```
typedef struct tftp_info_
{
  uint32 tftp_client_ip_addr;
  uint32 tftp_client_subnet_mask;
  uint32 tftp_default_gateway;
  uint32 tftp_server_ip_addr;
} tftp_info_st;

typedef struct hfr_location_
{
  int rack;
  int slot;
  int instance;
} hfr_location_st;
```

This defines the RSI location in a "global, universal" format.

```
typedef struct mbivec_
{
  char *mbi_name;
  size_t mbi_len;
} mbi_vec_st;
```

This is one "vector", in style of 'iov', to an MBI filespec.

```
typedef struct mbi_files_
{
   int num_cand;
   mbi_vec_st mbi_list[0];
} mbi_files_st;
```

The above is what a client feeds to MBI API: how many names I'm giving you, then the names.

mbimgr_validate_mbivsn:

Inputs: card_type
hw_rev
location
mbi_files
Outputs: ret_filespec
tftp_info

```
cerrno mbimgr_validate_mbivsn (int card_type, ulong hw_rev,
                    hfr_location_st * location,
                    mbi_files_st * mbi_files,
                    char** ret_filespec,
                    tftp_info_st * tftp_info);
```

The return code alone can specify the caller reaction, as long as the API behaviour is well defined. Possible values are:

MBIMGR_ERROR_SUCCESS the returned 'ret_filespec' filename vsn is the one to boot; tftp_info is valid if filespec device == "tftp:" (fields will be non-zero)

General errors, caller would likely retry:

MBIMGR_ERROR_INVALIDARG something distasteful in the args
MBIMGR_ERROR_NOMEM needed to but couldn't alloc mem
MBIMGR_ERROR_NOSERVER could not contact the MBI server process

The struct pointed to by 'tftp_info' will always be filled in. All zeroes means that tftp info is not needed, ie. the indicated filespec is not to be sourced from tftp.

The 'card_type' value is unique to each type of LC (and SP/DRP/LC). There may be a more official 'type' than 'int' in the future. 'hw_rev' is associated with the card in question, is unique to that manufactured card.

'location' (ie. RSI) is needed since the card might be part of a particular LR and could potentially use a different MBI than the same card type in another LR.

In addition, a user can configure image sets for a given node uniquely.

'ret_filespec' is a returned as a pointer to a malloc'd buffer, and is a complete file spec, such as

bootflash:/zone1/drp_0_6_2/drp_mbi.vm

tftp:/sc/images/pos48/pos48_0_5_1/lc_mbi.vm

etc

If there was an error, ret_filespec will be set to NULL. If non-NULL, it must be free'd by the caller to avoid a memory leak.

The caller provides 'tftp_info', a ptr to a tftp_info_st structure. In the case of returning a filespec for a file to be tftp'd (ie. prepended with "tftp:"), the tftp info is filled in. Otherwise, the struct is zeroed.

Note: The device string 'bootflash:' may or may not be sent up from rommon, depending upon implementation and future directions. MbiMgr will adapt to the final implementation.

The intention of this API is to review N candidate input MBI file specs, and hand out "acceptance" of one MBI version, ie. that the indicated filespec is "acceptable" to use. This does not imply that the given filespec is the latest or greatest MBI version, only that, according to whatever rules & policy are contained within and behind MbiMgr, it is acceptable to boot the card using the given MBI.

Upon success, the filespec returned may be one of the files given as input, a NULL pointer, or a new one (obtained from a tftp server).

5.0 Internal Specifications

There is currently no existing API for MbiMgr to use to query for appropriate MBI versioning for a given node.

It has been agreed that InstallMgr must maintain the association of node and card_type per LR to MBI and install stream. It is not deemed useful for MbiMgr to replicate this information.

This API has been proposed:

cerrno libinst_select_mbi(char* node_name, char* card_type,
  char *local_mbi_names, int len, char* selected_mbi_name, int out_len)

[Issue #6 - prefer to use scalars for node_name, card_type]

6.0 Packaging Considerations

MbiMgr, as a component, will be added to a software package such that it is available for use on all SC platforms.

It should not be installed onto any other card type.

7.0 End User Interface

MbiMgr will provide a debug command to log all client request activity on this SC:

debug mbimgr

Besides "debug" commands, the following config-mode commands are hidden and intended for internal use only.

In configuration mode:

boot location <RSI> <image-filespec> [<tftp-ssrvr-info>]

sets the image to be loaded for a specific node. There is no plan to check the validity of the indicated image for the card type which may be present when image loading is invoked. Note that the user must still issue a 'reload' command after this config setting in order for the desired image to be loaded.

boot card-type <card-type-value> <image-filespec> [<tftp-ssrvr-info>]

Same as above, but the setting applies to all of the indicated card-types within the scope of the MbiMgr. If the card is non-SC, then it applies to all such cards within the rack. If an SC card is indicated, then the command will be effective only if the CLI node is the dSC.

boot [location] local-mbi [disable]

This setting tells MbiMgr to decline the validation request for any local (to bootflash) MBI, and return instead a filespec for the remotely stored MBI file (to be tftp booted). This setting is enabled by default; ie. local MBI validations are always considered.

8.0 Testing Considerations

To ensure that MbiMgr is operating correctly, it is sufficient to exercise the installation of images using Install Manager to various cards, and ensuring that the correct images are booted. Turning on 'debug mbimgr' will show the correct MBI's being validated.

In addition, the configuration override commands listed in section 7.0 can be exercised to ensure that developers can exert a measure of control over the boot procedure.

9.0 RAS Considerations

The RAS considerations, which stands for "Reliability, Availability, Serviceability," will be addressed in the Design Specification for this feature.

10.0 Issues, Risks, and Dependencies

10.1 Dependencies

MbiMgr is dependent upon Install Manager to store and retrieve all information required to validate any card's MBI.

MbiMgr is dependent upon configuration data being available for boot image overrides. This dependency is treated as non-critical: if the config data cannot be sourced, then normal processing will be used.

10.2 Open Issues

Issue #1

When SC rommon cannot locate a dSC to validate one of its MBI's, which one does it choose to boot?

First answer: if a SYSPAGE entry with the name of an MBI which was previously booted exists, and matches a local MBI, then it can be booted.

Otherwise, there is no rule yet established.

Issue #2

Requirements placed upon rommon:

- rommon needs to write the (MBI) image name booted to SYSPAGE
- rommon needs to write the "boot server source" to SYSPAGE
- for SP/LC/DRP: rommon needs to write the local rack number to SYSPAGE

For SP/LC/DRP, the boot server source is simply the SC slot number. For an SC, it is the dSC's rack and slot numbers (if dSC was contacted, else something invalid).

Issue #3

Can sysmgr operate correctly without QSM, GSP, and Sysdb?

Issue #4

A new qnet development may require that an "Active Open" be performed for an ENA node to become visible on the qnet network. This will require that PCtl or dSCp directly opens some well-known server on the node's (remote) MBI in order that its qnet become usable. (The well-known server would be contained within the MBI-Hello process). The intention is to avoid the scalability issues involved with qnet node broadcasting.

If "Active Open" works bidirectionally, then it would be sufficient for a subservient MBI process to "open" a channel to the SC that it knows must exist. rommon has written the 'boot server source' to SYSPAGE, so Install Helper knows where to find it. For example, a qnet path such as these might be opened:

/net/rack4/sc_slot0/slot0/<my_dir>/<my_rev>/<files>

/net/rack5/sc_slot1/dev/install_mgr

etc

Issue #5

According to reference [4], Install Helper requires Version Manager API's and Sysdb acccess. We must therefore package these entities (client-side API's only) into the MBI. At the same time, these libraries are subject to change and upgrade, especially for "regular" components. Install Helper itself will not likely be able to use any such upgraded API's, so the versions bundled into the MBI must suffice until a complete MBI upgrade is needed, requiring a node reboot. In addition, QSM and GSP are not available for any server access. Qnet must suffice to perform the required Install Helper duties.

Issue #6

Investigate modification of the proposed InstallMgr API:

cerrno libinst_select_mbi(char* node_name, char* card_type,

char *local_mbi_names, int len, char* selected_mbi_name, int out_len)

Appendix A References

As appropriate, detail other documents which have bearing on the activities detailed in this document.

Documents referenced from the text:

1. ENG-114800 HFR Designated System Controller dSC Functional Specification
2. ENG-118962 HFR Plugin Controller: Software Unit Design Specification
3. ENG-102416 Card Configuration Protocol Specification
4. ENG-126996 Distributed Software Manager Design Specification
5. ENG-83268 HFR Install Distribution: Software Unit Functional Specification
6. ENG-79408 HFR Software Image Management: Architecture Overview

Other relevant documents:

ENG-96459 IOS/ENA Install Manager: Software GSR FCS Functional Specification

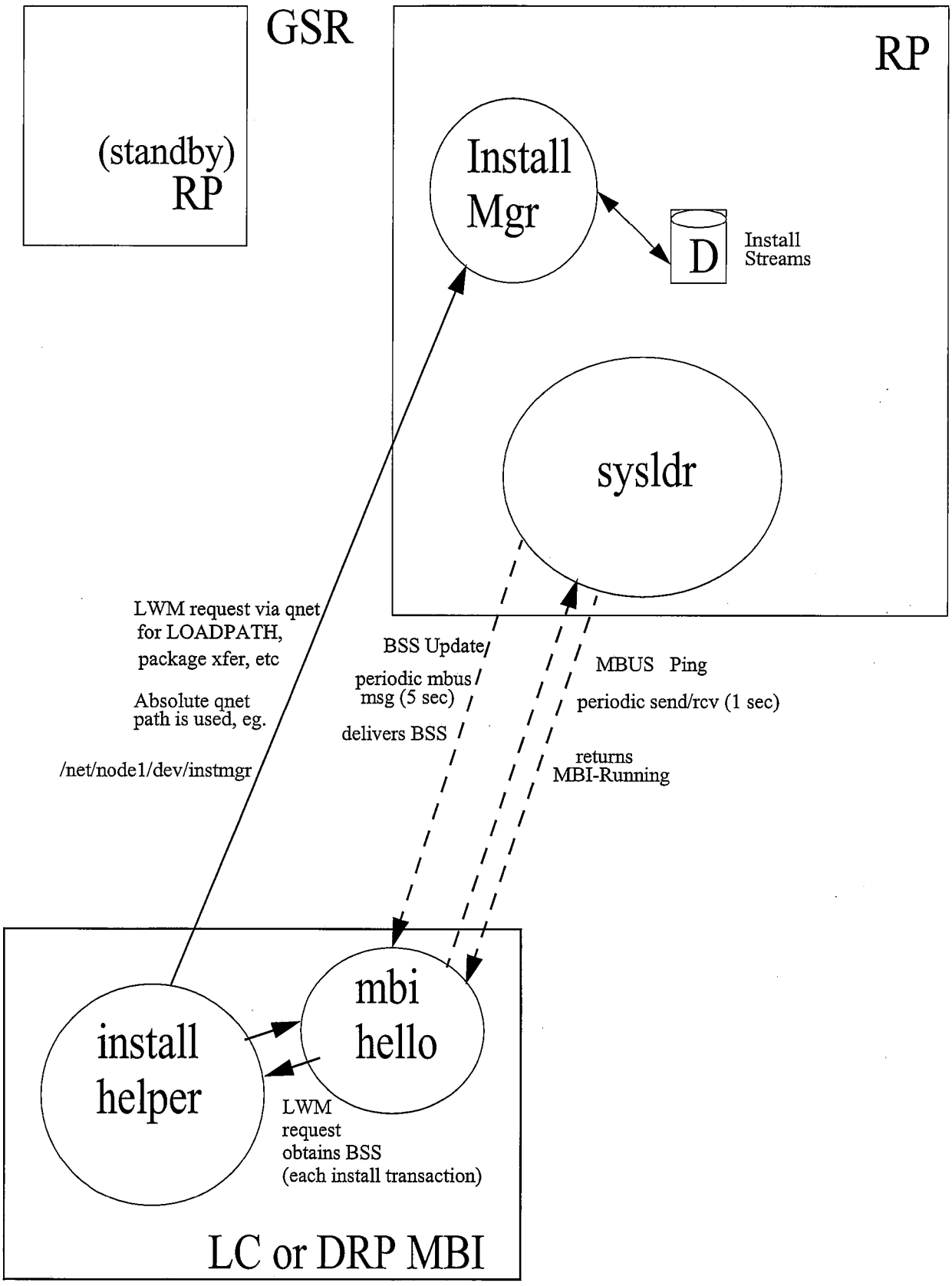
ENG-96575 HFR Configurations Housekeeping: Architecture Overview

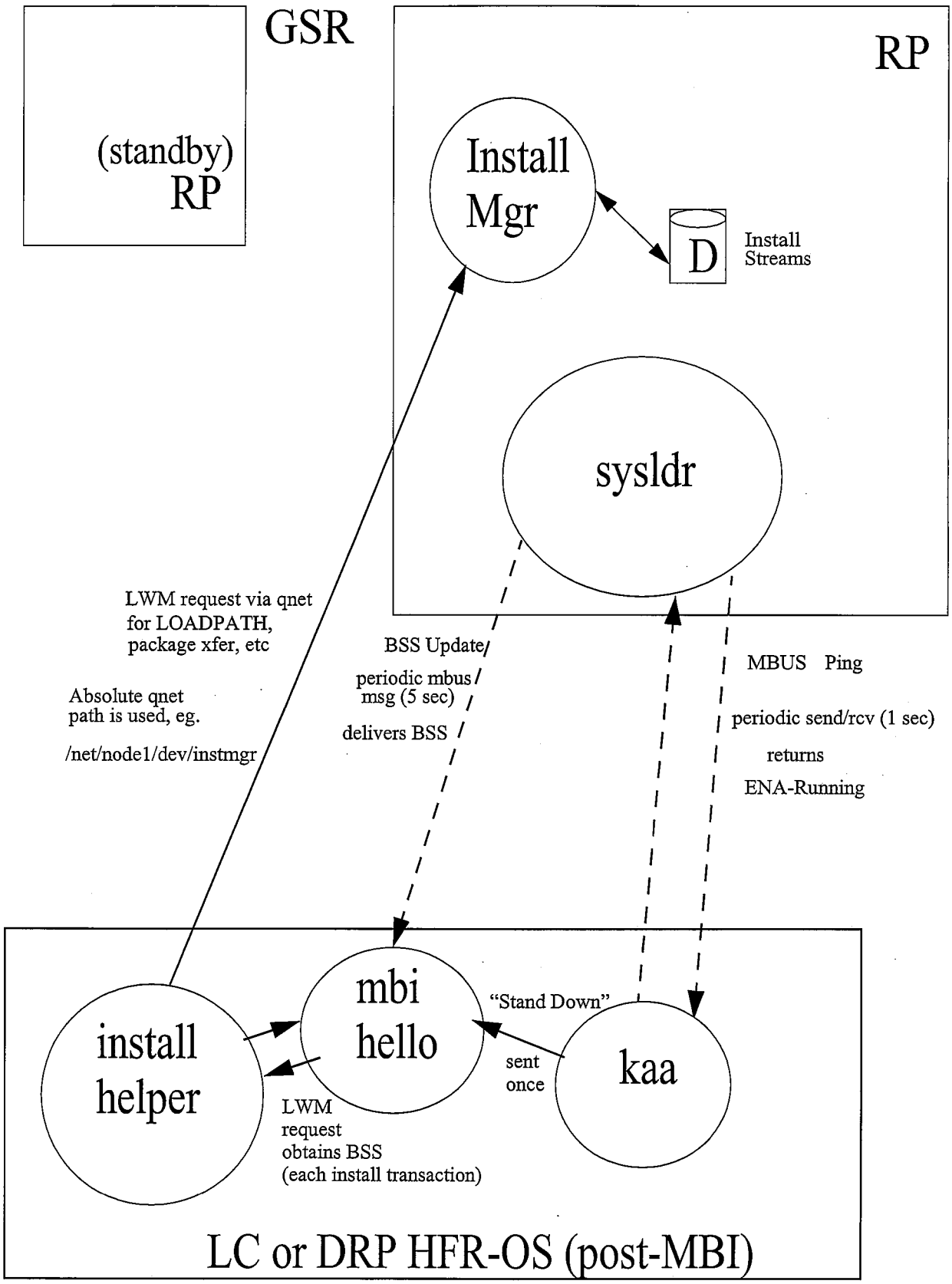
ENG-82685 HFR: System Controller Architecture

ENG-108775 HFR Shelf Controller Platform Manager: Software Functional Specification

ENG-106984 QSM

ENG-128878 GSP


GSR
(standby)
RP
RP
Install
Mgr
D
Install
Streams
sysldr
LWM request via qnet
for LOADPATH,
package xfer, etc
Absolute qnet
path is used, eg.
/net/node1/dev/instmgr
BSS Update/
periodic mbus
msg (5 sec)
delivers BSS
MBUS Ping
periodic send/rcv (1 sec)
returns
MBI-Running
mbi
hello
install
helper
LWM
request
obtains BSS
(each install transaction)
LC or DRP MBI


GSR
(standby)
RP
RP
Install
Mgr
D
Install
Streams
sysldr
LWM request via qnet
for LOADPATH,
package xfer, etc
Absolute qnet
path is used, eg.
/net/node1/dev/instmgr
BSS Update/
periodic mbus
msg (5 sec)
delivers BSS
MBUS Ping
periodic send/rcv (1 sec)
returns
ENA-Running
mbi
hello
"Stand Down"
sent
once
kaa
install
helper
LWM
request
obtains BSS
(each install transaction)
LC or DRP HFR-OS (post-MBI)

11.0 MBI-Hello Operation

11.1 GSR Functionality

mbi-hello is a persistent process which is part of the OS package, ie. part of the MBI itself. It is used to assist Install Helper in locating Install Manager. The Install Manager executes on a node known as the Boot Server Source (BSS).

Inside the MBI world, there are no QSM services, so no symlink lookup for /dev/instmgr is possible. Instead, insthelper must discover the BSS node and "manually" construct an absolute qnet path in order to connect to it.

This is done by having mbi-hello receive mbus frames sent from sysldr indicating the BSS. mbi-hello then remembers it for whenever a client, such as insthelper, might wish to know. The client interface is via a library call, 'mbi_get_boot_server_node()', which uses an LWM exchange.

insthelper contacts installmgr for each transaction using 'msg_chan_connect()'. This avoids having to maintain an open connection for rare future 'upgrade' events. It picks up the BSS from mbi-hello before each connection attempt, constructs the qnet path to try. For a given transaction, such as a package pull, insthelper keeps trying a potentially "new" BSS until connection succeeds.

On the primary RP node, where there is an implicit assumption that Install Manager runs there as BSS, the sysldr process periodically sends the BSS indicator frames to mbi-hello. These are unicasted to each running node, both during MBI phase and forever afterwards at intervals.

BSS is updated by sysldr into mbi-hello in the background. Normally, this sends the same BSS value as the last update. This continuous update covers the case of dropped messages. It is possible for BSS to change. One example of when this might happen is after an RP failover. The new sysldr on the new primary RP updates all mbi-hello processes on all served nodes with the new BSS.

No attempt is made by insthelper to use qsm symlink to locate instmgr, even during post-MBI phase.

11.2 GSR scenarios covered

insthelper retries: Each transaction, such as a package pull, is preceded by an attempt in a loop to connect to instmgr. If it fails, a new copy of the BSS is fetched for the next retry.

RP failover: sysldr on new primary RP sends down new BSS node to each mbi-hello process. During failover, no install transactions can succeed, since mbi-hello will still point to old BSS until failover is complete.

mbi-hello restart: Upon restart, mbi-hello assumes 'unknown' for BSS. A sysldr thread sends the BSS every 5 seconds via mbus to each mbi-hello process (one per node) to keep it updated.

HFR-OS handoff: The "handoff" from running in the MBI phase to HFR-OS phase is achieved by having the Keep Alive Agent (KAA), present in the Base package, tell mbi-hello to "stand down" via an LWM message. This instructs mbi-hello to cease answering the mbus ping. This allows KAA to answer the ping uniquely, thereby indicating to sysldr that HFR-OS is running.

11.3 HFR Functionality

mbi-hello is a persistent process which is part of the OS package, ie. part of the MBI itself. It is used to assist Install Helper in locating Install Manager. The Install Manager executes on a node known as the Boot Server Source (BSS).

Inside the MBI world, there are no QSM services, so no symlink lookup for /dev/instmgr is possible. Instead, insthelper must discover the BSS node and "manually" construct an absolute qnet path in order to connect to it.

This is done by having mbi-hello receive ethernet frames sent from shelfmgr indicating the BSS. mbi-hello then remembers it for whenever a client, such as insthelper, might wish to know. The client interface is via a library call, 'mbi_get_boot_server_node()', which uses an LWM exchange.

The message sending the BSS also sends a real-time time stamp. The first one received by mbi-hello is used to set the local node's real-time clock. This is so that any syslog messages generated during the MBI phase will have the correct time stamp. Once the Base package is pulled and started, ntp services will keep the local clock up to date.

insthelper contacts installmgr for each transaction using 'msg_chan_connect()'. This avoids having to maintain an open connection for rare future 'upgrade' events. The function performing the connect has a loop with a timeout. At the top of the loop, an attempt is made to use the QSM symlink /dev/instmgr to connect. If it fails - which it will during MBI phase - it next picks up the BSS from mbi-hello, constructs the qnet path, and tries the connection attempt again. For a given transaction, such as a package pull, insthelper keeps trying, first, the symlink, then a potentially "new" BSS until connection succeeds, or a timeout occurs

On the primary RP node, the shelfmgr process periodically sends the BSS indicator frames to mbi-hello during the MBI phase (only). These are unicasted to each running node during its MBI phase at intervals.

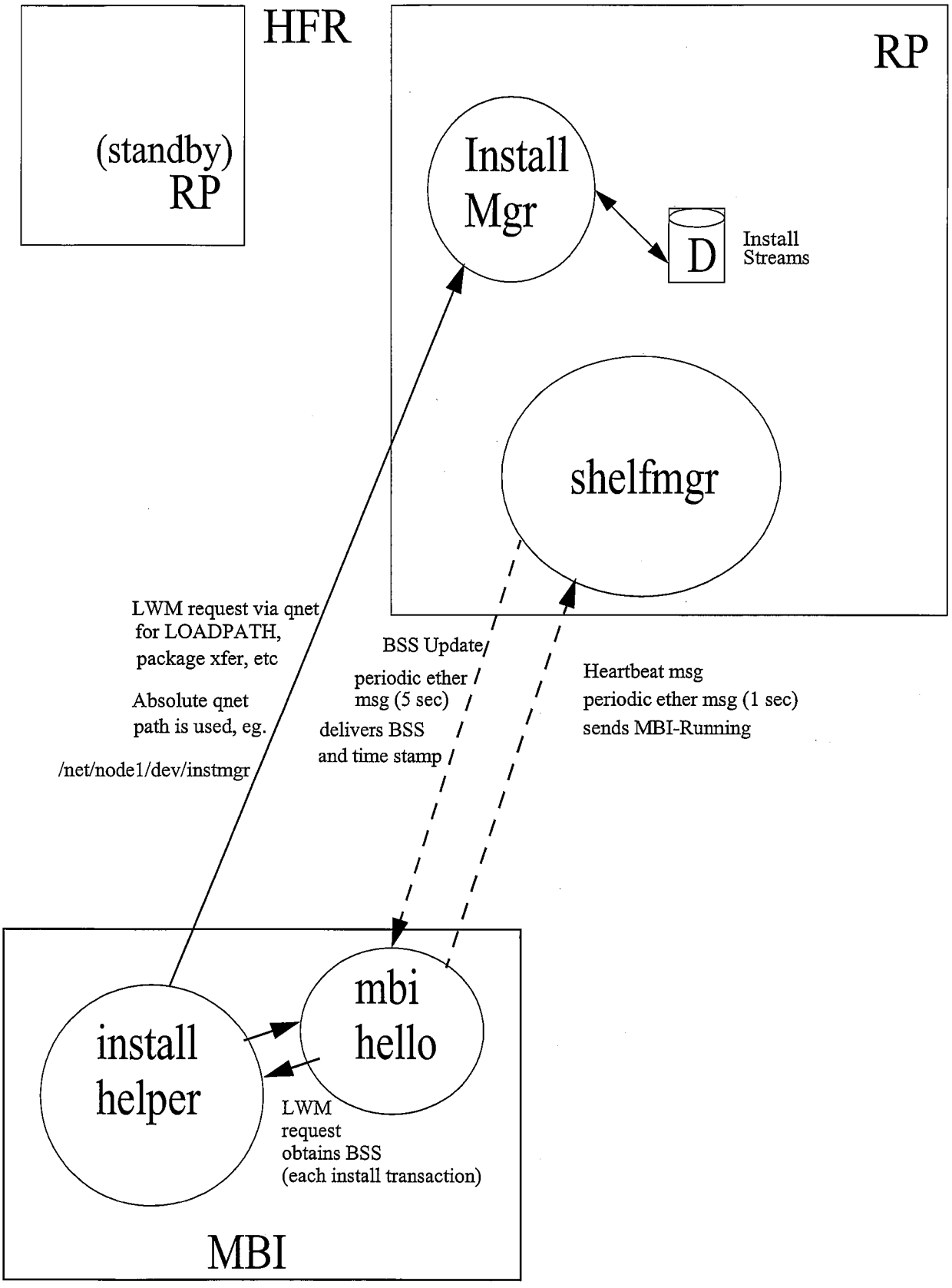
11.4 HFR scenarios covered

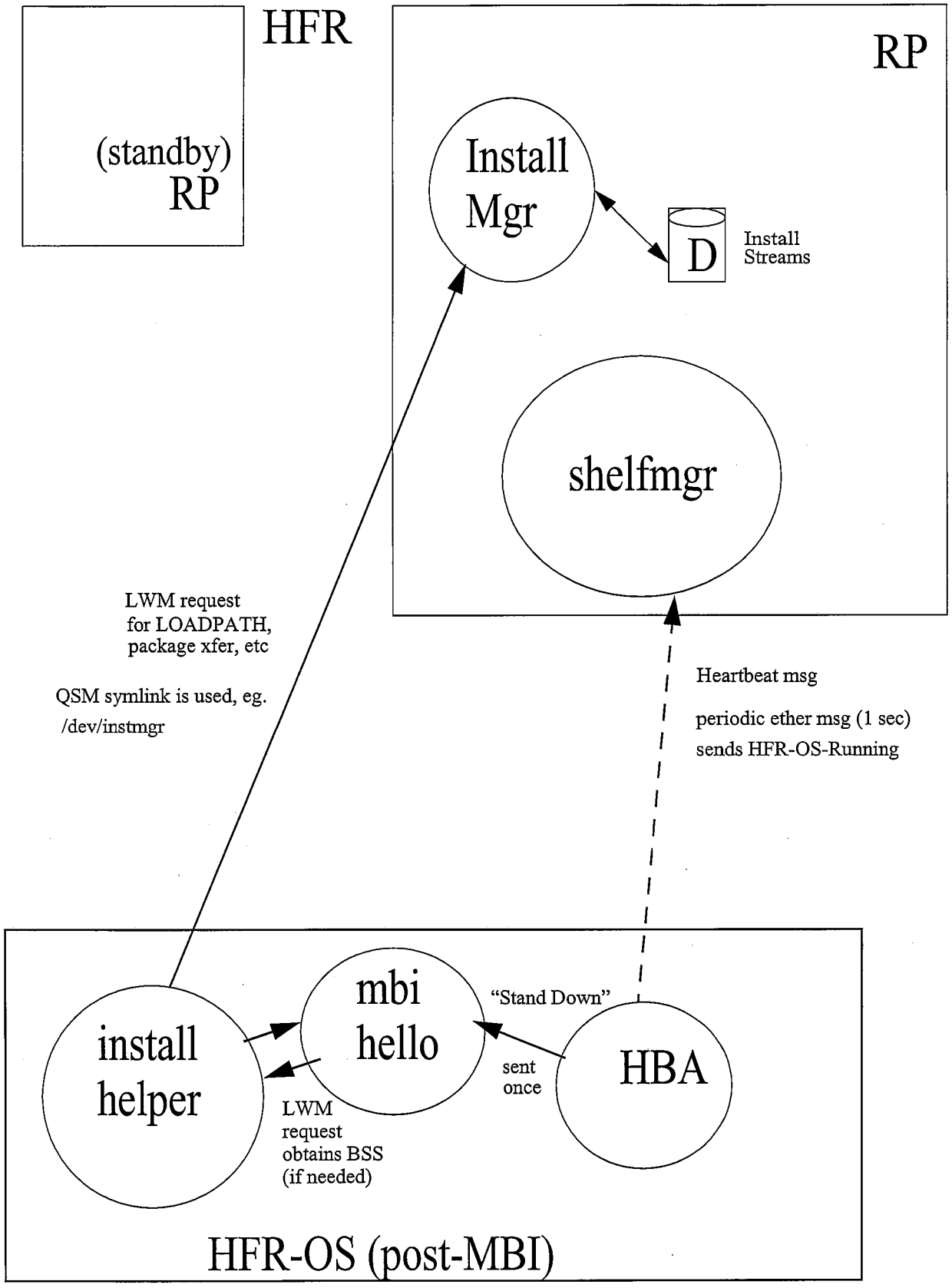
insthelper retries: Each transaction, such as a package pull, is preceded by an attempt in a loop to connect to instmgr. The loop tries first the QSM symlink, then a potentially new copy of the BSS is fetched for the next retry.

RP failover: shelfmgr on new primary RP sends down new BSS node to each mbi-hello process. During failover, no install transactions can succeed, since mbi-hello will still point to old BSS until failover is complete.

mbi-hello restart: Upon restart, mbi-hello assumes 'unknown' for BSS. shelfmgr sends the BSS every 5 seconds via ethernet to each mbi-hello process (one per node) to keep it updated.

HFR-OS handoff: The "handoff" from running in the MBI phase to HFR-OS phase is achieved by having the Heartbeat Agent (HBA), present in the Base package, tell mbi-hello to "stand down" via an LWM message. This instructs mbi-hello to cease sending heartbeat messages. This allows HBA to send heartbeats uniquely, thereby indicating to shelfmgr that HFR-OS is running.


HFR
(standby)
RP
RP
Install
Mgr
D
Install
Streams
shelfmgr
LWM request via qnet
for LOADPATH,
package xfer, etc
Absolute qnet
path is used, eg.
/net/node1/dev/instmgr
BSS Update/
periodic ether
msg (5 sec)
delivers BSS
and time stamp
Heartbeat msg
periodic ether msg (1 sec)
sends MBI-Running
mbi
hello
install
helper
LWM
request
obtains BSS
(each install transaction)
MBI


HFR
(standby)
RP
RP
Install
Mgr
D
Install
Streams
shelfmgr
LWM request
for LOADPATH,
package xfer, etc
QSM symlink is used, eg.
/dev/instmgr
Heartbeat msg
periodic ether msg (1 sec)
sends HFR-OS-Running
install
helper
mbi
hello
"Stand Down"
sent
once
HBA
LWM
request
obtains BSS
(if needed)
HFR-OS (post-MBI)